

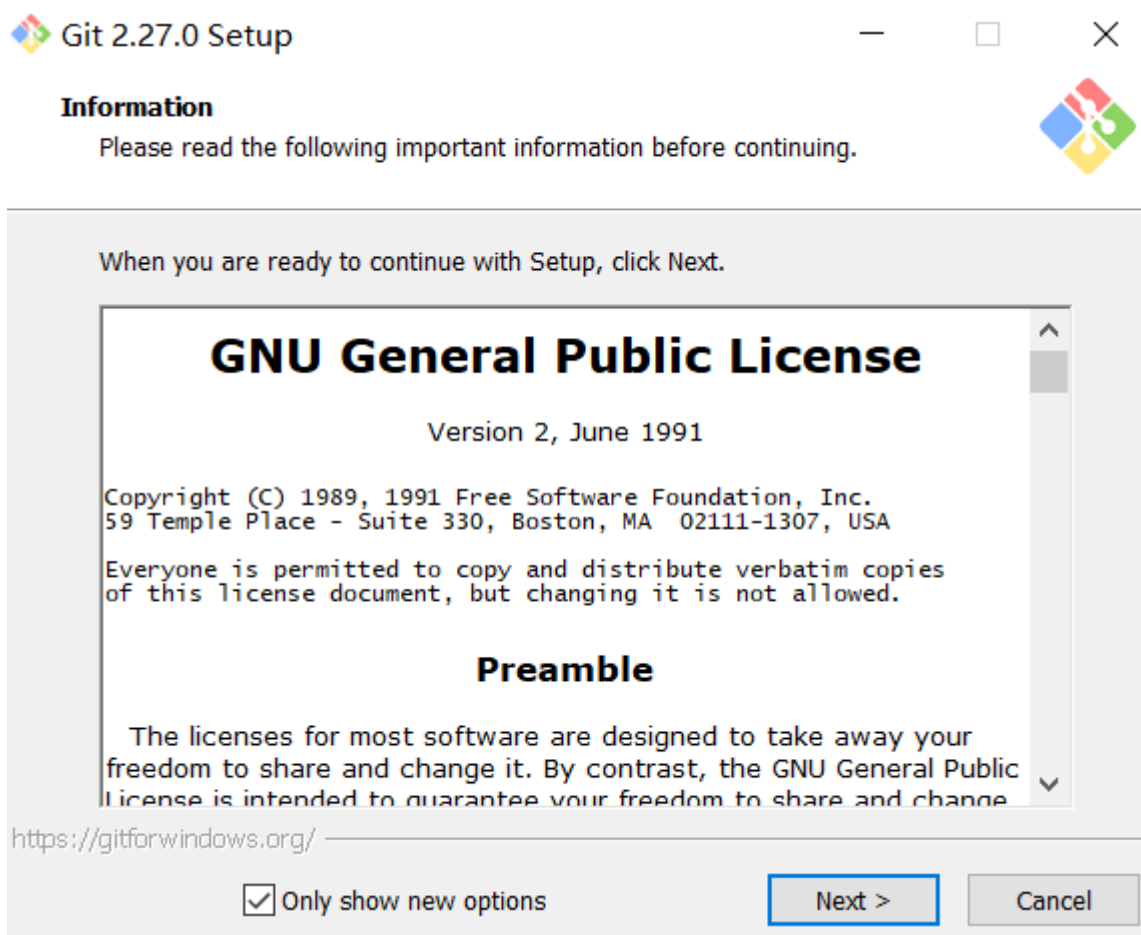
马士兵教育Python 全栈文档

第二章：咚宝商城项目-项目创建

1、创建远程仓库

1、下载和安装GIT

<https://git-scm.com/download/win>



2、创建远程仓库

登陆注册Github<https://github.com/> 或者登录注册码云<https://gitee.com/>。操作步骤是一样的，考虑到以后我们使用码云速度会快很多，所以我们在项目中用码云gitee。



新建仓库

仓库名称 ✓

dong_mall

归属



msb-goldbin

路径 ✓

/ dong_mall

仓库地址: https://gitee.com/msb-goldbin/dong_mall

仓库地址很重要

仓库介绍 非必填

马士兵教育Python全栈之Flask项目

仓库描述

是否开源



私有



公开

私有只能最多5人开发

任何人都可以访问该仓库的代码和其他任何形式的资源

选择语言

Python

添加 .gitignore

Python

添加开源许可证 ⓘ

点此快速选择许可证

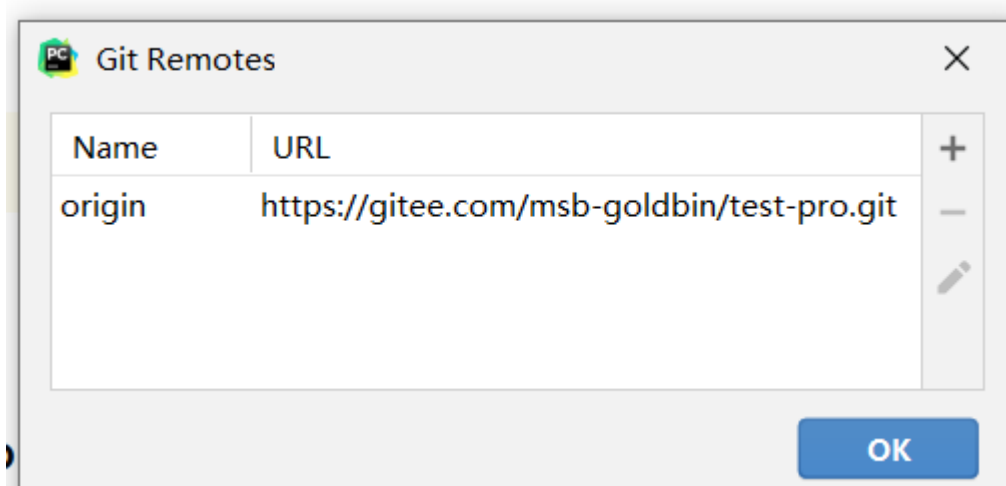
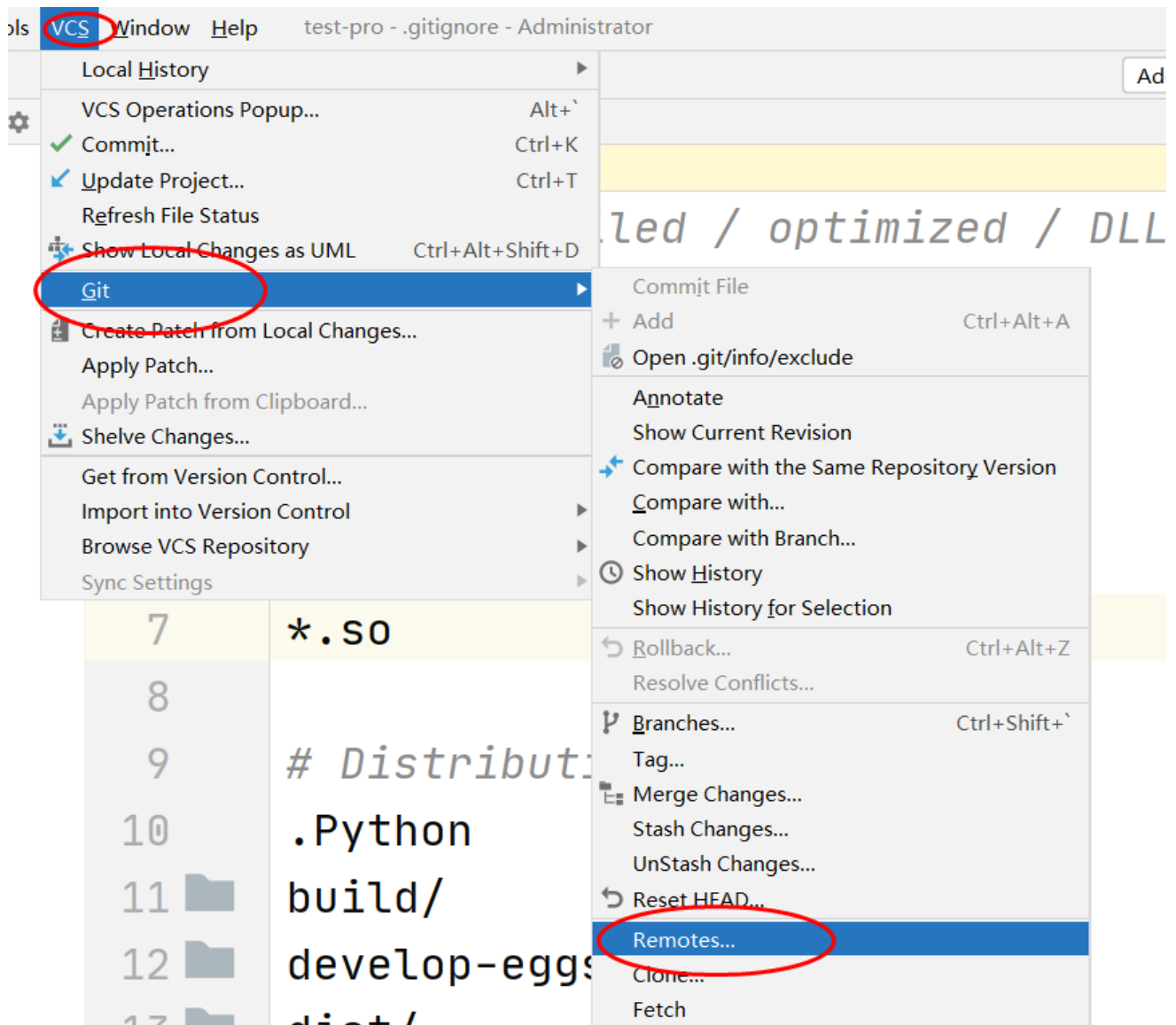
请选择开源许可证

☐ 使用Readme文件初始化这个仓库

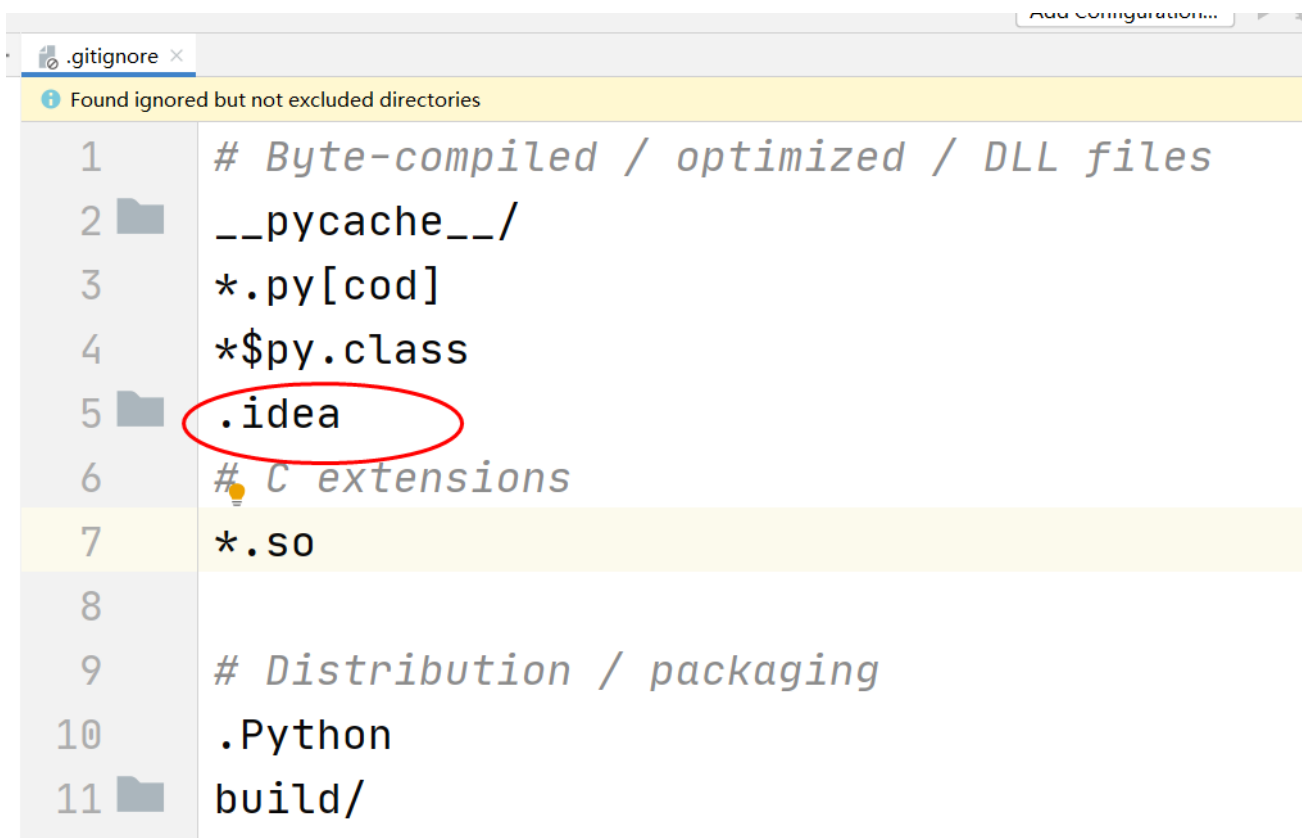
3、通过命令克隆到本地仓库

```
C:\Users\Administrator>workon flask_env
(flask_env) C:\Users\Administrator>d:
(flask_env) D:\env\flask_env>cd ../../
(flask_env) D:\>cd PycharmProjects
(flask_env) D:\PycharmProjects>git clone https://gitee.com/msb-goldbin/dong_mall.git^X
```

使用Pycharm打开本地仓库



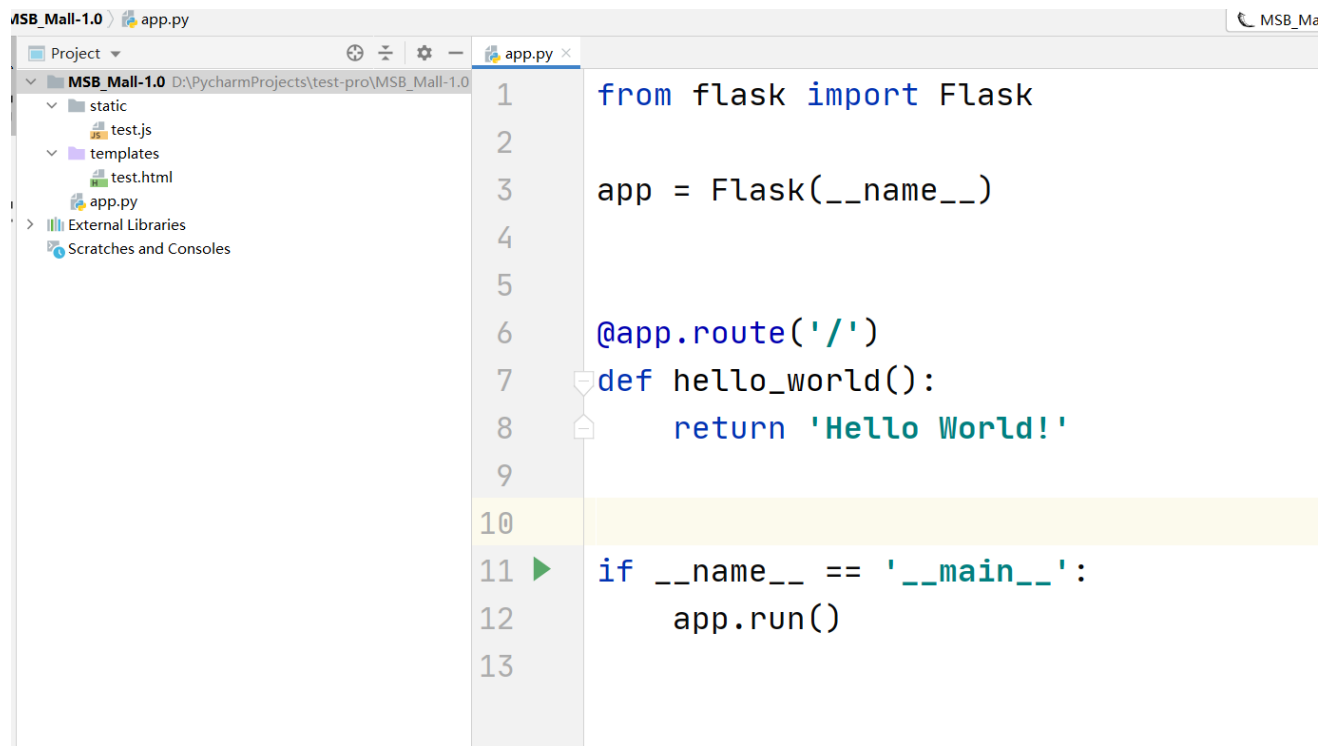
修改忽略文件：增加.idea配置。可以限制Pycharm软件生成的文件不提交到仓库中。



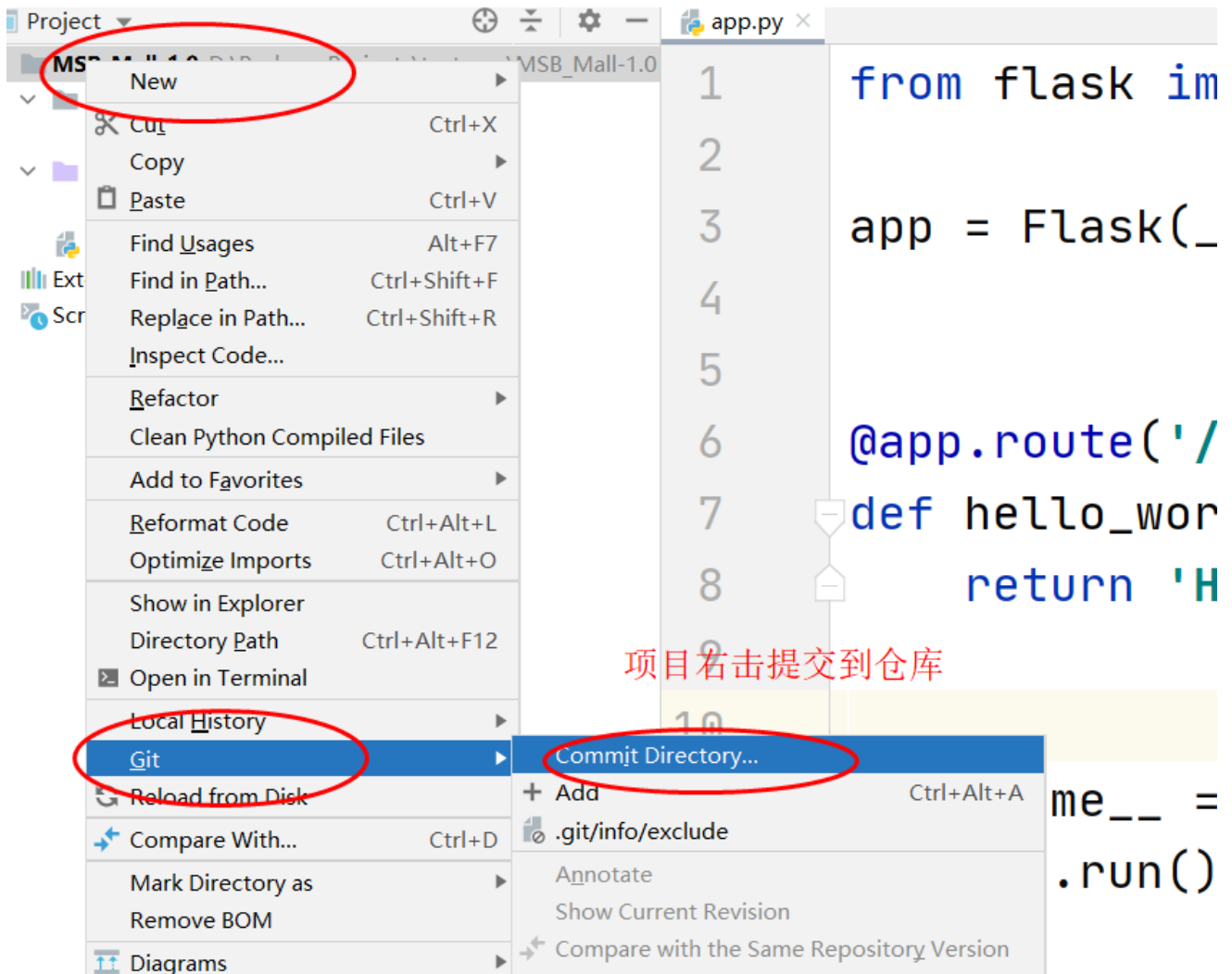
```
.gitignore
Found ignored but not excluded directories
1 # Byte-compiled / optimized / DLL files
2 __pycache__ /
3 *.py[cod]
4 *$py.class
5 .idea
6 # C extensions
7 *.so
8
9 # Distribution / packaging
10 .Python
11 build/
```

4、创建项目

首先新建一个新的Python虚拟环境，然后安装Flask



```
MSB_Mall-1.0 > app.py
Project
MSB_Mall-1.0 D:\PycharmProjects\test-pro\MSB_Mall-1.0
  static
  test.js
  templates
  test.html
  app.py
  External Libraries
  Scratches and Consoles
1 from flask import Flask
2
3 app = Flask(__name__)
4
5
6 @app.route('/')
7 def hello_world():
8     return 'Hello World!'
9
10
11 if __name__ == '__main__':
12     app.run()
13
```



2、Flask-Script

1、Flask-Script介绍

Flask-Script的作用是可以通命令行的形式来操作Flask。

Flask Script扩展提供向Flask插入外部脚本的功能，包括运行一个开发用的服务器，一个定制的Python shell，设置数据库的脚本，cronjobs，及其他运行在web应用之外的命令行任务；使得脚本和系统分开；

```
pip install flask-script 安装
```

2、Flask-Script基本使用

在一个在flask项目中，新建一个新的hello.py，其中的hello功能函数我们希望通过命令来运行。

```
from flask_script import Manager
from momo import app

# 第一步
manager = Manager(app)

#第二步
#1. 通过命令执行
```

```
@manager.command
def hello():
    print('你好,hello')

if __name__ == '__main__':
    # 第三步: 启动manager
    manager.run()
```

在命令行中执行:

```
if __name__ == '__main__':

Terminal: Local < +

(flask_env) D:\PycharmProjects\test-pro\MSB_Mall-1.0>python hello.py hello
你好,hello

(flask_env) D:\PycharmProjects\test-pro\MSB_Mall-1.0>
```

3、Flask-Script案例

需求: 通过命令直接在数据库中创建一个用户

数据库连接配置config.py文件

```
HOSTNAME = '127.0.0.1'
PORT = '3306'
DATABASE = 'test'
USERNAME = 'root'
PASSWORD = '123123'

DB_URI = "mysql+pymysql://{username}:{password}@{host}:{port}/{db}?
charset=utf8".format(username=USERNAME,password=PASSWORD,host=HOSTNAME,port=PORT,db=DAT
ABASE)
SQLALCHEMY_DATABASE_URI = DB_URI
SQLALCHEMY_TRACK_MODIFICATIONS = False
```

主程序app.py

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
import config
app = Flask(__name__)
app.config.from_object(config)
db = SQLAlchemy(app)

class User(db.Model):
    __tablename__ = 't_user'
    id = db.Column(db.Integer,primary_key=True,autoincrement=True)
    uname = db.Column(db.String(50),nullable=False)
    pwd = db.Column(db.String(50),nullable=False)
    age = db.Column(db.Integer,nullable=True)

# db.create_all()
```

```
@app.route('/')
def hello_world():
    return 'Hello world!'

if __name__ == '__main__':
    app.run()
```

执行命令的addUserTest.py

```
from flask_script import Manager
from app import app, User, db

manager = Manager(app)

#3.通过命令快速建立一个后台账号
@manager.option("-u", "--uname", dest="uname")
@manager.option("-p", "--password", dest="pwd")
def add_user(uname, pwd):
    user = User(uname=uname, pwd=pwd)
    db.session.add(user)
    db.session.commit()
    print("添加OK")

if __name__ == '__main__':
    manager.run()
```

执行命令添加用户:

```
(flask_env) D:\PycharmProjects>cd D:\PycharmProjects\test-pro\MSB_Mall-1.0
(flask_env) D:\PycharmProjects\test-pro\MSB_Mall-1.0>python addUserTest.py add_user -u laoxiao -p 123123
添加OK
(flask_env) D:\PycharmProjects\test-pro\MSB_Mall-1.0>
(flask_env) D:\PycharmProjects\test-pro\MSB_Mall-1.0>
```

3、Flask-Migrate数据库模型映射

1、Flask-Migrate介绍

flask-migrate可以十分方便的进行数据库的迁移与映射，将我们修改过的ORM模型映射到数据库中。flask-migrate是基于Alembic进行的一个封装，并集成到Flask中，所有的迁移操作其实都是Alembic做的，他能跟踪模型的变化，并将变化映射到数据库中。

```
pip install flask-migrate
```

2、Flask-Migrate使用

在项目中新建一个db_manager.py

```
from flask_script import Manager
from app import app
```



```
from exts import db

from flask_migrate import Migrate,MigrateCommand

#需要把映射到数据库中的模型导入到manage.py文件中
from models import User
manager = Manager(app)

#用来绑定app和db到flask-migrate的
Migrate(app,db)
#添加Migrate的所有子命令到db下
manager.add_command("db",MigrateCommand)

if __name__ == '__main__':
    manager.run()
```

3、Flask-Migrate命令

1. 初始化一个环境: `python db_manage.py db init`
2. 自动检测模型, 生成迁移脚本: `python db_manage.py db migrate`
3. 将迁移脚本映射到数据库中: `python db_manage.py db upgrade`
4. 更多命令: `python db_manage.py db --help`

4、建立模型

1、配置数据库参数

2、编写模型与映射

3、项目模块结构

5、项目日志配置

用Python写代码的时候, 在想看的地方写个print 就能在控制台上显示打印信息, 这样子就能知道它是什么了, 但是当我需要看大量的地方或者在一个文件中查看的时候, 这时候print就不大方便了, 所以Python引入了logging模块来记录我想要的信息。

主要包括四部分:

- Logger: 可供程序直接调用的接口, app通过调用提供的api来记录日志
- Handlers: 决定将日志记录分配至正确的目的地
- Filters: 对日志信息进行过滤, 提供更细粒度的日志是否输出的判断
- Formatters: 制定最终记录打印的格式布局

1、logger

loggers 就是程序可以直接调用的一个日志接口，可以直接向logger写入日志信息。logger并不是直接实例化使用的，而是通过logging.getLogger(name)来获取对象，事实上logger对象是单例模式，logging是多线程安全的，也就是无论程序中哪里需要打日志获取到的logger对象都是同一个。

2、Handlers

Handlers 将logger发过来的信息进行准确地分配，送往正确的地方。举个栗子，送往控制台或者文件或者both或者其他地方(进程管道之类的)。它决定了每个日志的行为，是之后需要配置的重点区域。

每个Handler同样有一个日志级别，一个logger可以拥有多个handler也就是说logger可以根据不同的日志级别将日志传递给不同的handler。当然也可以相同的级别传递给多个handlers这就根据需求来灵活的设置了。

3、Filters

Filters 提供了更细粒度的判断，来决定日志是否需要打印。原则上handler获得一个日志就必定会根据级别被统一处理，但是如果handler拥有一个Filter可以对日志进行额外的处理和判断。

4、Formatters

Formatters 指定了最终某条记录打印的格式布局。Formatter会将传递来的信息拼接成一条具体的字符串，默认情况下Format只会将信息%(message)s直接打印出来。Format中有一些自带的LogRecord属性可以使用，如下表格：

Attribute	Format	Description
asctime	%(asctime)s	将日志的时间构造成可读的形式，默认情况下是'2016-02-08 12:00:00,123'精确到毫秒
filename	%(filename)s	包含path的文件名
funcName	%(funcName)s	由哪个function发出的log
levelname	%(levelname)s	日志的最终等级（被filter修改后的）
message	%(message)s	日志信息
lineno	%(lineno)d	当前日志的行号
pathname	%(pathname)s	完整路径
process	%(process)s	当前进程
thread	%(thread)s	当前线程

一个Handler只能拥有一个Formatter 因此如果要实现多种格式的输出只能用多个Handler来实现。

5、日志等级

- **等级：DEBUG < INFO < WARNING < ERROR < CRITICAL**
- **DEBUG**：最详细的日志信息，主要的应用场景问题的诊断，只限于开发人员使用的，用来在开发过程中进行调试
- **INFO**：详细程度仅次于debug模式，主要来记录关键节点的信息，确定程序是否正常如预期完成，一般的使用场景是重要的业务处理已经结束，我们通过这些INFO级别的日志信息，可以很快的了解应用正在做什么。
- **WARNING**：当某些不被期望的事情发生的时候，需要记录的信息，比如磁盘即将存满，注意当前的程序一依旧可以正常运行，不报错。也就是说发生这个级别的问题时，处理过程可以继续，但必须要对这个问题给予额外的关注。
- **ERROR**：出现严重问题，导致某些功能不能正常运行记录信息
- **CRITICAL**：系统即将崩溃或者已经崩溃